Overcoming the Pitfalls of Prediction Error in Operator Learning for Bilevel Planning

Nishanth Kumar[1], Willie McClinton[1], Tomás Lozano-Pérez, Leslie Kaelbling
MIT CSAIL
{njk,wbm3,tslvr,tlp,lpk}@csail.mit.edu

***Abstract*—Bilevel planning, in which a high-level search over an abstraction is used to guide low-level decision-making, is an effective approach to solving long-horizon tasks in continuous state and action spaces. Recent work has shown how to enable such bilevel planning by learning action and transition model abstractions in the form of symbolic operators and neural samplers. In this work, we show that existing symbolic operator learning approaches fall short in many robotics environments where agent actions tend to cause a large number of irrelevant propositions to change. This is primarily because they attempt to learn operators that optimize the prediction error with respect to observed changes in the propositions. To overcome this issue, we propose to learn operators that only model changes *necessary* for abstract planning to achieve the specified goal. Experimentally, we show that our approach learns operators that lead to efficient planning across 10 different hybrid robotics domains, including 4 from the challenging BEHAVIOR-100 benchmark, with generalization to novel initial states, goals, and objects.**

I. Introduction

Solving long-horizon robotics problems in domains with continuous state and action spaces is extremely challenging, even when the transition function is deterministic and known. One effective recipe is to learn abstractions for the domain and then perform hierarchical planning to any new goal. A typical approach is to first learn state abstractions in the form of symbolic *predicates* (classifiers on the low-level state, such as `InGripper`), then learn *operator descriptions* and *samplers* in terms of these predicates [37, 11]. The operators describe a partial transition model in the abstract space, while the samplers enable the search for realizations of abstract actions in terms of primitive actions. In this paper, we focus on the problem of learning operator descriptions from a very small set of demonstrations given a set of predicates, an accurate low-level transition model, and a set of parameterized controllers (such as `Pick(x, y, z)`) that serve as primitive actions. By learning such operators, we hope to leverage bilevel planning to aggressively generalize to a highly variable set of problem domain sizes, initial states, and goals in challenging robotics domains.

A natural objective for the problem of finding good abstract domain models is minimizing prediction error [35, 37]. This objective would be appropriate if we were using the abstract model to make predictions, but in fact we are using it as *planning guidance* for an accurate low-level model. So instead, our objective is to find an abstract model that maximally improves the performance of the planning algorithm, given the available data. The difference between these objectives is stark in many natural domains where each action taken by the agent can change a large number of propositions. To make highly accurate predictions about all of these state changes might require a very fine-grained model with many complex operators. Such a model would require a lot of data to learn reliably, be very slow to plan with, and also be unlikely to generalize to novel tasks with different numbers and configurations of objects.

For example, consider the "Sorting Books" task (Figure 1) from the recent BEHAVIOR-100 benchmark [38]. Here, the agent's goal is to retrieve a number of books strewn about a living room and place them on a particular shelf. A `Reachable(?object)` predicate is given to indicate when the agent is close enough to a particular object to pick it up. When the agent moves to pick a particular object, the set of objects that are reachable varies depending on the specific configuration of objects. For instance, the figure shows a transition where the agent moves to put itself in range of picking up the particular book `book7`, but happens to also be in range of picking up 1 other book and 3 other items of different types. Optimizing prediction error on this transition would yield a rather complex operator such as the one below:

Op−MoveToBook−Prediction−Error:
Args: ?objA ?objB ?objC ?objD ?objE ?objF ?objG
Preconditions: (**and** (Reachable ?objA) (Reachable ?objB))
Add Effects: (**and** (Reachable ?objC) (Reachable ?objD) (Reachable ?objE) (Reachable ?objF) (Reachable ?objG))
Delete Effects: (**and** (Reachable ?objA) (Reachable ?objB))

This operator is overfit to this specific situation and thus neither useful for efficient high-level planning nor generalizable to new tasks with different object configurations (e.g. the test situation depicted in the right panel of Figure 1, where the robot is not in reachable range of any objects).

In this work, we observe that, in order to generate useful high-level plans, *operators need only model changes in predicates that are necessary for high-level search*. Through learning operators to only model these necessary changes, we enable better generalization and faster planning. For example, we can learn the operator below, and use it to plan to reach a target object while *ignoring*, through delete effects, the reachability of all other objects (note that this operator generalizes to the test situation in the right panel of Figure 1).

[1]Equal Contribution

Fig. 1. **Example demonstration transition and evaluation task from BEHAVIOR-100**. (Left) Visualization and high-level states for an example transition where the agent moves from being in range of picking a number of objects. (Right) Visualization of an evaluation task where the agent starts out not in range of picking any objects.

Op–MoveToBook–Planning–Performance:
Args: ?objA
Preconditions: ()
Add Effects: (Reachable ?objA)
Delete Effects: ($\forall$?x. ?objA $\neq$?x $\Rightarrow$ (Reachable ?x))

Our main contributions are (1) the formulation of a new objective for learning an abstract model, (2) a procedure for distinguishing necessary changes within the high-level states of provided demonstrations, and (3) an algorithm that leverages (1) and (2) to learn symbolic operators from demonstrations via a hill-climbing search. We test our method on a wide range of complex robotic planning problems and find that our learned operators enable bilevel planning to solve challenging tasks and generalize substantially from a small number of examples. This is particularly noteworthy in two of the more complex environments from the BEHAVIOR-100 set [38], since no other method we know of is able to achieve non-negligible test performance in these tasks.

II. Background

A. Problem Setting

We consider the problem of learning operators for search-then-sample bilevel planning [35, 11]. In this setting, a state $x \in \mathcal{X}$ is characterized by the continuous properties (e.g. pose, color, material) of a set of objects. Objects may be typed (e.g., `robot`, `book`) and properties can vary between types. A set of *predicates* defines discrete properties of objects (e.g., `HandEmpty`) or relations between objects (e.g., `On`). Predicates induce a state abstraction $\textsc{abstract} : \mathcal{X} \rightarrow \mathcal{S}$ where $\textsc{abstract}(x)$ is the set of true *ground atoms* in x (e.g., {`HandEmpty(robot)`, `On(b1, b2)`, ...}). An action is a hybrid *controller* $u \in \mathcal{U}$, which has discrete and continuous parameters (e.g., `Pick(block,` θ`)` where θ is a continuous grasp). Transitions are deterministic and a simulator $f : \mathcal{X} \times \mathcal{U} \rightarrow \mathcal{X}$ predicts the next state given a current state and action. The state space, predicates, action space, and simulator comprise an *environment*.

A *task* $T \in \mathcal{T}$ is characterized by a set of objects $\mathcal{O}$, an initial state $x_0 \in \mathcal{X}$, and a goal g. The goal is a set of ground atoms and is *achieved* in x if $g \subseteq \textsc{abstract}(x)$. A *solution* to a task is a sequence of actions $\overline{u} = (u_1, \ldots, u_n)$ that achieve the goal ($g \subseteq \textsc{abstract}(x_n)$, and $x_i = f(x_{i-1}, u_i)$ for $1 \leq i \leq n$). Each environment is associated with a *task distribution*. Our objective is to maximize the likelihood of solving tasks from this distribution within a planning time budget. We consider learning-based approaches that use a set of *training tasks* $\mathcal{T}_{\text{train}} \subset \mathcal{T}$. Each training task is given with one demonstration, i.e., a solution for the task. Collectively, these demonstrations make a set $\mathcal{D}$ that we use during learning. After training, we test generalization to a set of evaluation tasks also drawn from $\mathcal{T}$.

B. Planning Operators

In bilevel planning (see Appendix VII-A for more details), search in the abstract state space $\mathcal{S}$ is used to guide planning in the low-level state space $\mathcal{X}$. The abstract search requires *operators*. Importantly, note that unlike in classical planning, *our operators need not yield downward refinable plans*.

An operator ω has *arguments* $\overline{v}$, *preconditions* P, *add effects* E^+, *delete effects* E^-, and a *controller* C. The arguments are placeholders for objects. The preconditions and effects are each expressions over the arguments (more on this below). The controller associates some of the arguments with its discrete parameters and does not have continuous parameters specified (e.g., `Pick(?v,·)`). A substitution of arguments to objects induces a *ground operator* $\underline{\omega} = \langle \underline{P}, \underline{E^+}, \underline{E^-}, \underline{C} \rangle$ where the placeholders are substituted accordingly.

We consider operators that can be expressed in a particular subset of PDDL [19], which allows us to leverage efficient AI planning techniques for search. Following previous work [35, 11], we represent operator preconditions as a set of atoms (i.e., a logical conjunction). Ground preconditions $\underline{P}$ are satisfied in s if $\underline{P} \subseteq s$. Add effects are also represented with a set of atoms and ground add effects $\underline{E^+}$ are added to s via set union. In addition to the typical atomic delete effects, we also permit universally quantified single-predicate delete effects (e.g., $\forall$`?v. Reachable(?v)`). We consider this more expressive class of delete effects because of our desire to move away from prediction error and instead focus on goal reachability: these quantified effects allow us to "ignore more" of the abstract state. During grounding, these delete effects are transformed into ground atoms $\underline{E^-}$ by substituting the quantified variables with each object in the state. Altogether, given a ground operator $\underline{\omega} = \langle \underline{P}, \underline{E^+}, \underline{E^-}, \underline{C} \rangle$, if $\underline{P} \subseteq s$, then the successor abstract state s' is $(s \setminus \underline{E^-}) \cup \underline{E^+}$ where $\setminus$ is set difference. We use $F(s, \underline{\omega}) = s'$ to denote this (partial) abstract transition function.

In previous work on operator learning for bilevel planning [35, 11], operators are connected to the underlying

environment through the following semantics: if $F(s, \underline{\omega}) = s'$, then *there exists* some low-level transition (x, u, x') where $\textsc{abstract}(x) = s$, $\textsc{abstract}(x') = s'$, and $u = \underline{C}(\theta)$ for some θ. These semantics embody the "prediction error" view in that ABSTRACT must predict the entire next state ($\textsc{abstract}(x') = s'$). Towards implementing the alternative "necessary changes" view, we will instead only require the abstract state output by F to be a *subset* of the atoms in the next state, that is, $\textsc{abstract}(x') \subseteq s'$. Intuitively, F is now responsible for generating abstract *subgoals* to guide low-level planning, rather than predicting entire successor abstract states.

III. Learning Operators from Demonstrations

The efficiency and effectiveness of bilevel planning is highly dependent on the planning operators' ability to provide *useful guidance* for low-level planning [37]. The prediction error objective used in prior work (e.g., [35]) is *misaligned* with this objective because it incentivizes learned operators to be highly specific at the expense of generating refinable high-level plans. Optimizing prediction error forces operators to predict the entire abstract state correctly. In many natural domains, especially those with a large number of objects, executing a particular controller is likely to result in a number of *irrelevant* changes. Accurately predicting the abstract state in such domains leads to a large number of learned operators that are each hyper-specific (they have a large number of preconditions and effects). Consequently, these operators generate plans that make hyper-specific predictions for each abstract state that are unlikely to hold true, especially in evaluation tasks that have different numbers of objects in different configurations than the training tasks.

Intuitively, it seems unnecessary and even wasteful to model *all* observed changes. Indeed, we only need model the few changes that are *necessary* for generating refinable plans. We now formalize this intuition.

A. Necessary Atoms

Definition III.1. Given an abstract plan $(\underline{\omega}_1, \ldots, \underline{\omega}_n)$ that achieves goal g, the *necessary atoms* at step n are $\alpha_n \triangleq g$, and at step $0 \leq i < n$ are $\alpha_i \triangleq \underline{P_{i+1}} \cup (\alpha_{i+1} \setminus \underline{E^+_{i+1}})$.

In other words, the necessary atoms are the conditions required for the remaining abstract plan to be feasible. Each necessary atom set is minimal in that no atoms can be removed without violating either the goal or a future operator's preconditions. This property creates an appealing basis for operator learning: if the necessary atoms are correct, then the remainder of the plan must be feasible (at the abstract level). Furthermore, in many robotic domains, the necessary atoms will be a small subset of the full abstract state. We thus endeavor to learn operators that predict the necessary atoms instead of the entire next abstract state.

However, there is an unfortunate circularity lurking in the definition above: to identify necessary atoms, one must *already have* operators. In other words, it is not possible to examine the training demonstrations, immediately determine necessary atom sets for each step, and then do supervised learning. We will remedy this by conducting a local search over operator sets, and use the *current candidate* operator set to compute necessary atoms.

B. Hill-Climbing Search over Operator Sets

We perform a hill-climbing search over operator sets, starting with the empty set. To implement this search, we must define (1) an objective function to minimize, and (2) a mechanism for proposing successor candidate operator sets. Both of these components will use the current candidate operator set to (partially) compute necessary atoms for the demonstrations via a type of goal-regression in the form of *preimage backchaining* [33, 31, 42] (see Appendix for more details). We now describe these components in more detail.

1) Hill-Climbing Objective: Our objective consists of two terms: a *coverage* term, which penalizes operator sets that are unable to yield plans that mimic the demonstrations for training tasks (Definition VII.1 in the Appendix); and a *complexity* term, which provides regularization by preferring simpler operator sets. Formally, we want to minimize:

$$J(\Omega) \triangleq \texttt{coverage}(\mathcal{D}, \Omega) + \lambda \texttt{complexity}(\Omega) \quad (1)$$

where $\lambda > 0$ is a small constant that trades off the relative importance of the terms. Intuitively, we compute `coverage` by stepping backwards from the goal and counting how many transitions cannot be matched (or 'covered') some existing operator (i.e, the operator's precondition is true in the initial state, the effects hold in the next state, etc.) in the current candidate set for each demonstration (See Appendix VII-C1 for details). Complete coverage (i.e, `coverage` = 0) means that we will be able to reproduce all the demonstration action sequences on all demonstration tasks using our learned operators. We implement `complexity` simply by counting the number of operators (i.e, $\texttt{complexity}(\Omega) \triangleq |\Omega|$).

2) Generating Successor Operator Sets: We propose hill-climbing search over operator sets with two successor generators (see Appendix VII-C1 and VII-C2 for more details). The first and main successor generator `improve-coverage` uses preimage backchaining to find a necessary atoms set that is currently uncovered, and then proposes a set of new operators to cover it. The second successor generator `reduce-complexity` simply deletes operators from the current candidate set, which can help to prevent the proliferation of unnecessary operators. Note that the `improve-coverage` generator is guaranteed to decrease the `coverage` cost term of our objective in Equation III-B1, while `reduce-complexity` is guaranteed to decrease the `complexity` cost term. In our experiments, we set the constant λ in the objective to be small enough such that improving the `coverage` term always improves the overall objective more than improving the `complexity` term.

IV. Experiments

A. Experimental Setup

We evaluate six methods across ten robotic planning environments of varying difficulty. Four of our baselines (LOFT,

Environment	Ours	LOFT	LOFT+Replay	CI	CI + QE	GNN Shoot	GNN MF
Painting	**98.80 (1.33)**	0.00 (0.00)	**98.20 (2.89)**	**99.00 (1.00)**	**93.40 (4.65)**	36.00 (10.73)	0.60 (0.92)
Satellites Simple	**93.40 (11.14)**	0.00 (0.00)	34 (16.71)	**91.60 (8.48)**	**95.20 (4.12)**	40.40 (9.62)	11.00 (4.58)
Cluttered 1D	**100.00 (0.00)**	17.20 (17.28)	0.00 (0.00)	17.40 (17.46)	92.80 (2.86)	98.60 (2.01)	98.60 (2.01)
Screws	**100.00 (0.00)**	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	50.00 (50.00)	95.60 (9.67)	95.80 (9.73)
Satellites	**95.20 (2.40)**	0.00 (0.00)	0.00 (0.00)	1.60 (1.96)	6.00 (4.98)	4.80 (4.02)	0.00 (0.00)
Cluttered Painting	**99.20 (1.33)**	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	4.60 (3.69)	0.00 (0.00)
Opening Presents	**100.00 (0.00)**	0.00 (0.00)	-	83.00 (34.07)	83.00 (34.07)	28.00 (18.87)	0.00 (0.00)
Locking Windows	**100.00 (0.00)**	0.00 (0.00)	-	90.00 (14.14)	88.00 (14.00)	0.00 (0.00)	0.00 (0.00)
Collecting Cans	**77.00 (37.16)**	0.00 (0.00)	-	0.00 (0.00)	1.00 (3.00)	0.00 (0.00)	0.00 (0.00)
Sorting Books	**69.00 (36.73)**	0.00 (0.00)	-	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)

TABLE I
PERCENTAGE SUCCESS RATE ON TEST TASKS FOR ALL DOMAINS. THE PERCENTAGE STANDARD DEVIATION IS SHOWN IN BRACKETS.

LOFT+Replay, CI, CI + QE) are from prior work that learn operators by optimizing prediction error, while two (GNN Shoot and GNN MF) learn an end-to-end policy from the demonstrations. Five environments were created by us, one was taken from Silver et al. [35], and four are from the BEHAVIOR-100 benchmark [38]. For the non-BEHAVIOR environments, we run all methods on 50 training demonstrations; for BEHAVIOR environments, we use 10, since collecting training data in these BEHAVIOR is very time and memory intensive.

All results are averaged over 10 random seeds. For each seed, we sample a set of *evaluation tasks* from the task distribution $\mathcal{T}$. *The evaluation tasks have more objects, different initial states, and goals with more atoms than seen during training.* Our key measure of effective bilevel planning is success rate within a timeout (10 seconds for non-BEHAVIOR environments, 500 for 3 BEHAVIOR environments, and 1500 for "Sorting Books": the most complex environment). For environment and baseline descriptions, as well as additional analyses, including the complexity of learned operators, examples of learned operators, and learning and planning efficiency, see Appendix VII-H1 and VII-I.

B. Results and Analysis

As seen in Table I, our method solves many more held-out tasks within the timeout than the baselines. In our simpler environments (Painting and Satellites Simple; Opening Presents and Locking Windows for BEHAVIOR), the controllers cause a small number of changes in the abstract state, and baseline approaches that optimize prediction error (CI, CI + QE) perform reasonably well. In all the other environments, the controllers cause a large number of changes in the abstract state, and the performance of operator learning baselines degrades substantially, though GNN baselines perform well on Cluttered 1D and Screws. Despite the increased complexity, our approach learns operators that enable bilevel planning to achieve a substantial test-time success rate under timeout. The performance in Collecting Cans and Sorting Books is especially notable; all baselines achieve a negligible success rate, while our approach achieves a near 70% rate on testing tasks. Upon investigation, we found that failures are due to local minima during hill-climbing for certain random seeds.

V. RELATED WORK

Our work continues a long line of research in learning operators for planning [14, 3, 23, 24, 26, 29, 32, 12, 2]; see Arora et al. [4] for a recent survey. Prior work generally focuses on learning operators from discrete plan traces in the context of classical (not bilevel) planning. An important difference between the the typical classical setting and ours is that our operators need not guarantee downward-refinability of the corresponding plans, since low-level geometric details can have a substantial effect on the feasibility of a high-level plan. Our focus is on learning a theory that makes the "global" bilevel planning as efficient as possible.

Other work has considered learning symbolic planning models in continuous environments [20, 39, 1, 5, 9, 6, 40, 22, 13]. Our efforts are most directly inspired by LOFT [35] and learning Neuro-Symbolic Relational Transition Models [11], which optimize prediction error to learn operators for bilevel planning. Like our method, LOFT performs a search over operator sets, but commits to modeling all effects seen in the data and searches only over operator preconditions. We point out the limitations of optimizing prediction error in complex environments. We include LOFT and Cluster and Intersect as baselines representative of these previous methods in our experiments.

Our work also contributes to a recent line of work on learning for TAMP. Other efforts in this line include sampler learning [10, 25, 41, 28], heuristic learning [34, 21, 30], and abstract plan feasibility estimation [15, 27].

VI. CONCLUSION AND FUTURE WORK

In this work, we proposed an objective for operator learning that is specifically tailored to bilevel planning, and a search-based method for optimizing this objective. Experiments confirmed that operators learned with our new method lead to substantially better generalization and planning than those learned by optimizing prediction error. Important next steps include integrating this method with predicate invention [37] and controller learning [36], as well as handling stochasticity and partial observability. We believe that pursuing these steps will yield important progress toward solving sparse-feedback, long-horizon decision-making problems at scale.

References

[1] Alper Ahmetoglu, M Yunus Seker, Justus Piater, Erhan Oztop, and Emre Ugur. Deepsym: Deep symbol generation and rule learning from unsupervised continuous robot interaction for planning. *arXiv preprint arXiv:2012.02532*, 2020.

[2] Diego Aineto, Sergio Jiménez, and Eva Onaindia. Learning strips action models with classical planning. In *The International Conference on Automated Planning and Scheduling (ICAPS)*, 2018.

[3] Eyal Amir and Allen Chang. Learning partially observable deterministic action models. *The Journal of Artificial Intelligence Research (JAIR)*, 2008.

[4] Ankuj Arora, Humbert Fiorino, Damien Pellier, Marc Métivier, and Sylvie Pesty. A review of learning planning action models. *The Knowledge Engineering Review*, 2018.

[5] Masataro Asai and Alex S. Fukunaga. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In *The AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[6] Masataro Asai and Christian Muise. Learning neural-symbolic descriptive planning models via cube-space priors: The voyage home (to STRIPS). *arXiv preprint arXiv:2004.12850*, 2020.

[7] Fahiem Bacchus. Aips 2000 planning competition: The fifth international conference on artificial intelligence planning and scheduling systems. *AI magazine*, 2001.

[8] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[9] Blai Bonet and Hector Geffner. Learning first-order symbolic representations for planning from the structure of the state space. In *The European Conference on Artificial Intelligence (ECAI)*, 2020.

[10] Rohan Chitnis, Dylan Hadfield-Menell, Abhishek Gupta, Siddharth Srivastava, Edward Groshev, Christopher Lin, and Pieter Abbeel. Guided search for task and motion plans using learned heuristics. In *The IEEE International Conference on Robotics and Automation (ICRA)*, 2016.

[11] Rohan Chitnis, Tom Silver, Joshua B. Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Learning neuro-symbolic relational transition models for bilevel planning. In *The IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2022.

[12] Stephen N Cresswell, Thomas L McCluskey, and Margaret M West. Acquiring planning domain models using LOCM. *The Knowledge Engineering Review*, 2013.

[13] Aidan Curtis, Tom Silver, Joshua B Tenenbaum, Tomás Lozano-Pérez, and Leslie Kaelbling. Discovering state and action abstractions for generalized task and motion planning. In *The AAAI Conference on Artificial Intelligence (AAAI)*, 2022.

[14] Gary L Drescher. *Made-up minds: a constructivist approach to artificial intelligence*. MIT press, 1991.

[15] Danny Driess, Jung-Su Ha, and Marc Toussaint. Deep visual reasoning: Learning to predict action sequences for task and motion planning from an initial scene image. In *Robotics: Science and Systems (R:SS)*, 2020.

[16] Caelan Reed Garrett, Rohan Chitnis, Rachel Holladay, Beomjoon Kim, Tom Silver, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Integrated task and motion planning. *Annual review of control, robotics, and autonomous systems*, 2021.

[17] Malte Helmert. The fast downward planning system. *The Journal of Artificial Intelligence Research (JAIR)*, 2006.

[18] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: what's the difference anyway? In *The International Conference on Automated Planning and Scheduling (ICAPS)*, 2009.

[19] Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins SRI, Anthony Barrett, Dave Christianson, et al. PDDL— the planning domain definition language. *Technical Report, Tech. Rep.*, 1998.

[20] Nikolay Jetchev, Tobias Lang, and Marc Toussaint. Learning grounded relational symbols from continuous data for abstract reasoning. In *ICRA Workshop on Autonomous Learning*, 2013.

[21] Beomjoon Kim and Luke Shimanuki. Learning value functions with relational state representations for guiding task-and-motion planning. *Conference on Robot Learning (CoRL)*, 2019.

[22] George Konidaris, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. From skills to symbols: Learning symbolic representations for abstract high-level planning. *The Journal of Artificial Intelligence Research (JAIR)*, 2018.

[23] Norbert Krüger, Christopher Geib, Justus Piater, Ronald Petrick, Mark Steedman, Florentin Wörgötter, Aleš Ude, Tamim Asfour, Dirk Kraft, Damir Omrčen, et al. Object–action complexes: Grounded abstractions of sensory–motor processes. *Robotics and Autonomous Systems*, 2011.

[24] Tobias Lang, Marc Toussaint, and Kristian Kersting. Exploration in relational domains for model-based reinforcement learning. *The Journal of Machine Learning Research (JMLR)*, 2012.

[25] Aditya Mandalika, Sanjiban Choudhury, Oren Salzman, and Siddhartha Srinivasa. Generalized lazy search for robot motion planning: Interleaving search and edge evaluation via event-based toggles. In *The International Conference on Automated Planning and Scheduling (ICAPS)*, 2019.

[26] Kira Mourao, Luke S Zettlemoyer, Ronald Petrick, and Mark Steedman. Learning strips operators from noisy and incomplete observations. *arXiv preprint arXiv:1210.4889*, 2012.

[27] Michael Noseworthy, Caris Moses, Isaiah Brand, Sebas-

tian Castro, Leslie Kaelbling, Tomás Lozano-Pérez, and Nicholas Roy. Active learning of abstract plan feasibility. In *Robotics: Science and Systems (R:SS)*, 2021.

[28] Joaquim Ortiz-Haro, Jung-Su Ha, Danny Driess, and Marc Toussaint. Structured deep generative models for sampling on constraint manifolds in sequential manipulation. In *Conference on Robot Learning (CoRL)*, 2022.

[29] Hanna M Pasula, Luke S Zettlemoyer, and Leslie Pack Kaelbling. Learning symbolic models of stochastic domains. *The Journal of Artificial Intelligence Research (JAIR)*, 2007.

[30] Chris Paxton, Vasumathi Raman, Gregory D Hager, and Marin Kobilarov. Combining neural networks and tree search for task and motion planning in challenging environments. In *The IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017.

[31] John L Pollock. The logical foundations of goal-regression planning in autonomous agents. *Artificial Intelligence*, 1998.

[32] Christophe Rodrigues, Pierre Gérard, Céline Rouveirol, and Henry Soldano. Active learning of relational action models. In *International Conference on Inductive Logic Programming*, 2011.

[33] Stanley J Rosenchein. Plan synthesis: A logical perspective. Technical report, SRI International Menlo Park CA Artificial Intelligence Center, 1981.

[34] William Shen, Felipe Trevizan, and Sylvie Thiébaux. Learning domain-independent planning heuristics with hypergraph networks. In *The International Conference on Automated Planning and Scheduling (ICAPS)*, 2020.

[35] Tom Silver, Rohan Chitnis, Joshua Tenenbaum, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Learning symbolic operators for task and motion planning. In *The IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021.

[36] Tom Silver, Ashay Athalye, Joshua B. Tenenbaum, Tomas Lozano-Perez, and Leslie Pack Kaelbling. Learning neuro-symbolic skills for bilevel planning. In *Conference on Robot Learning (CoRL)*, 2022.

[37] Tom Silver, Rohan Chitnis, Nishanth Kumar, Willie McClinton, Tomás Lozano-Pérez, Leslie Pack Kaelbling, and Joshua Tenenbaum. Predicate invention for bilevel planning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2023.

[38] Sanjana Srivastava, Chengshu Li, Michael Lingelbach, Roberto Martín-Martín, Fei Xia, Kent Elliott Vainio, Zheng Lian, Cem Gokmen, Shyamal Buch, Karen Liu, Silvio Savarese, Hyowon Gweon, Jiajun Wu, and Li Fei-Fei. BEHAVIOR: Benchmark for everyday household activities in virtual, interactive, and ecological environments. In *Conference on Robot Learning (CoRL)*, 2021.

[39] Emre Ugur and Justus Piater. Bottom-up learning of object categories, action effects and logical rules: From continuous manipulative exploration to symbolic planning. In *The IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[40] Elena Umili, Emanuele Antonioni, Francesco Riccio, Roberto Capobianco, Daniele Nardi, and Giuseppe De Giacomo. Learning a symbolic planning domain through the interaction with continuous environments. *ICAPS PRL Workshop*, 2021.

[41] Zi Wang, Caelan Reed Garrett, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Learning compositional models of robot skills for task and motion planning. *The International Journal of Robotics Research (IJRR)*, 2021.

[42] Daniel S. Weld. An introduction to least commitment planning. *AI Magazine*, 1994. doi: 10.1609/aimag.v15i4.1109. URL https://ojs.aaai.org/index.php/aimagazine/article/view/1109.

```
BILEVELPLANNING(O, x_0, g, Ψ, Ω, Σ)
    // Parameters: n_abstract, n_samples.
    s_0 ← ABSTRACT(x_0)
    // Outer Planning Loop
    for π̂ in GENABSPLAN(s_0, g, Ω, n_abstract) do
        // Inner Refinement Loop
        if REFINE(π̂, x_0, Ψ, Ω, n_samples) succeeds w/ π
            return π
```

Algorithm 1: Pseudocode for bilevel planning algorithm, adapted from Silver et al. [37]. The inputs are objects $\mathcal{O}$, initial state x_0, goal g, predicates Ψ, operators Ω, and samplers Σ; the output is a plan π. The outer loop GENABSPLAN generates high-level plans that guide our inner loop, which samples continuous parameters from our samplers Σ to concretize each abstract plan $\hat{\pi}$ into a plan π. If the inner loop succeeds, then the found plan π is returned as the solution; if it fails, then the outer GENABSTRACTPLAN continues.

VII. APPENDIX

A. Detailed Bilevel Planning Algorithm Descriptions

Given planning operators learned from training tasks $\mathcal{T}_{\text{train}}$, we can solve new evaluation tasks $T \in \mathcal{T}$ via search-then-sample bilevel planning (Algorithm 1). We provide a review of bilevel planning and refer to previous works for details [16, 11, 35, 36, 37]. Given the task goal g, initial state x_0, and corresponding abstract state $s_0 = \text{ABSTRACT}(x_0)$, bilevel planning uses efficient AI planning techniques (e.g., [17]) to generate candidate *abstract plans*. An abstract plan is a sequence of ground operators $(\underline{\omega}_1, \ldots, \underline{\omega}_n)$ where $g \subseteq s_n$ and $s_i = F(s_{i-1}, \underline{\omega}_i)$ for $1 \leq i \leq n$. The corresponding abstract state sequence $(s_1, \ldots, s_n)$ serves as a sequence of *subgoals* for low-level planning. Furthermore, the operator controller sequence $(\underline{C}_1, \ldots, \underline{C}_n)$ provides a *plan sketch*, where all that remains is to "fill in" the continuous parameters. We do this by sampling continuous parameters θ for each controller starting from the first and checking if the controller achieves its corresponding subgoal s_i. If we cannot sample such parameters within a constant budget, we backtrack and resample, eventually even generating a new abstract plan. Following previous work [11], we use learned neural network samplers to propose continuous controller parameters. See Appendix VII-E for more details.

Algorithms 2 and 3 provide pseudocode and additional details necessary for high-level and low-level search respectively in bilevel planning (Algorithm 1).

B. Preimage Backchaining

Given an operator set Ω and demonstration $(\overline{x}, \overline{u})$ for training task $T \in \mathcal{T}_{\text{train}}$ with goal g and objects $\mathcal{O}$, we use preimage backchaining (Algorithm 4) to compute an abstract plan *suffix* $(\underline{\omega}_k, \ldots, \underline{\omega}_n)$ (where $1 \leq k \leq n$) that is *consistent* with the demonstration.

```
GENABSTRACTPLAN(s_0, g, Ω, n_abstract)
    Ω̲ ← GROUND(Ω)
    // Search over ground operators from s0 to
       goal (returns top n plans).
    π̂ ← SEARCH(s_0, g, Ω̲, n_abstract)
    return π̂
```

Algorithm 2: This is GENABSTRACTPLAN which finds a high-level plan by creating operators for all possible groundings then uses search to find n_{abstract} plans. It returns a list of plans $\hat{\pi}$.

```
REFINE((π̂, x_0, Ψ, Σ, n_samples))
    state ← ABSTRACT(x_0)
    // While current state is not goal, sample
       and run current operator on current
       state and check ground atoms. If is
       passes continue if not backtrack.
    curr_idx ← 0
    while curr_idx < len(π̂) do
        samples[curr_idx] ← samples[curr_idx] + 1
        state_current, Ω̲ ← π̂[curr_idx]
        Ω̲.C.Θ ∼ Ω̲.Σ
        π[curr_idx] ← Ω̲.C
        curr_idx ← curr_idx + 1
        if Ω̲.C.initiable(state_current) then
            state_next ← Simulate(state_current, Ω̲.C)
            state_expected, _ ← π̂[curr_idx]
            if state_next ⊆ state_expected then
                can_continue_on ← True
                if curr_idx == len(skeleton) then
                    return sucess, π
            else
                canContinueOn ← False
        else
            canContinueOn ← False
        if not canContinueOn then
            curr_idx ← curr_idx − 1
            if samples[curr_idx] == max_samples
             then
                return failure, π
    return success, π
```

Algorithm 3: This is REFINE which turns a task plan $\hat{\pi}$ into a sequence of ground skills. It gets the state and operators from $\hat{\pi}$ and adds the controller with newly sampled continuous parameters to π. After this it checks to see if the added controller is initiable from the current state in the plan and we simulate the skill execution to verify it reached the expected state we predicted next in $\hat{\pi}$. If the controller is not initiable or fails the expected atoms check we backtrack and resample a new continuous parameter for this controller until either we reach the max number of samples or we successfully refine our final controller.

Definition VII.1. An abstract plan suffix $(\underline{\omega}_k, \ldots, \underline{\omega}_n)$ with necessary atoms $(\alpha_{k-1}, \ldots, \alpha_n)$ is *consistent* with a demon-

stration $(\overline{x}, \overline{u})$ for goal g and timesteps $1 \leq k \leq n$, where $\overline{x} = (x_0, \ldots, x_n)$ and $\overline{u} = (u_1, \ldots, u_n)$ if for $k \leq i \leq n$, (1) the states are consistent: $\alpha_i \subseteq F(\text{ABSTRACT}(x_{i-1}), \underline{\omega_i}) \subseteq \text{ABSTRACT}(x_i)$; and (2) the actions are consistent: if the controller for $\underline{\omega_i}$ is $\underline{C_i}$, then $u_i = \underline{C_i}(\theta)$ for some θ.

If the candidate operators Ω are sufficient to generate an abstract plan that mimics the demonstration provided for the task, then preimage backchaining (Algorithm 4) should yield a plan suffix of the same length. Otherwise, preimage backchaining will stall at some time step $k \geq 1$ where no operator can be used to reach necessary atoms α_k. This stalling is the main signal that we use to define our hill-climbing objective, and later to propose successor operator sets.

The pseudocode and description for preimage backchaining is shown in Algorithm 4.

PREIMAGEBACKCHAINING($(\Omega, (\overline{x}, \overline{u}), \mathcal{O}, g)$)

$n \leftarrow \text{length}(\overline{x})$
$\alpha_n \leftarrow g$
for $i \leftarrow n-1, n-2, \ldots, 0$ **do**
  $s_i, s_{i+1} \leftarrow \text{ABSTRACT}(x_i), \text{ABSTRACT}(x_{i+1})$
  $\underline{\omega}_{\text{best}} \leftarrow \text{FINDBESTCONSISTENTOP}(\Omega,\ s_i,\ s_{i+1},\ \alpha_{i+1},\ \mathcal{O})$
  if $\underline{\omega_{best}} = \textit{Null}$ **then**
    break
  $\underline{\omega_{i+1}} \leftarrow \underline{\omega}_{\text{best}}$
  $\alpha_i \leftarrow \underline{P_{i+1}} \cup (\alpha_{i+1} \setminus \underline{E^+_{i+1}})$
return $(\underline{\omega_{i+1}}, \ldots, \underline{\omega_n}), (\alpha_i, \ldots, \alpha_n)$

Subroutine FINDBESTCONSISTENTOP($(\Omega,\ s_i,\ s_{i+1},\ \alpha,\ \mathcal{O})$)

$\Omega_{\text{con}} \leftarrow \emptyset$
for $\omega \in \Omega$ **do**
  for $\underline{\omega} \in \text{GETALLGROUNDINGS}(\omega, \mathcal{O})$ **do**
    $s^{\text{pred}}_{i+1} \leftarrow ((s_i \setminus \underline{E^-}) \cup \underline{E^+})$
    if $\underline{P} \subseteq s_i$ *AND* $\alpha \subseteq s^{pred}_{i+1}$ *AND* $s^{pred}_{i+1} \subseteq s_{i+1}$ *AND* $\exists \theta : \underline{C}(\theta) = u_i$ **then**
      $\Omega_{\text{con}} \leftarrow \Omega_{\text{con}} \cup \underline{\omega}$
if $\Omega_{con} \neq \emptyset$ **then**
  return FINDBESTCOVER($\Omega_{con}, (s_i, s_{i+1})$)
return *Null*

Algorithm 4: Pseudocode for our preimage backchaining algorithm. The inputs are a set of operators Ω, a training demonstration $(\overline{x}, \overline{u})$, objects used in the corresponding task $\mathcal{O}$, and the task goal g. The output is an abstract plan suffix $(\underline{\omega_{i+1}}, \ldots, \underline{\omega_n})$ and corresponding sequence of necessary atoms $(\alpha_i, \ldots, \alpha_n)$. Intuitively, we pass backward through each transition in a trajectory while attempting to choose an operator in Ω to "cover" the transition and then updating the transition's necessary atoms using Definition III.1. If multiple such operators exist, we use a heuristic (FINDBESTCOVER) that attempts to select the one that best matches the transition (see Appendix VII-D for details).

C. Detailed Description of Successor Generators

IMPROVECOVERAGE(Ω, $\mathcal{D}$, $\mathcal{T}_{train}$)

$\text{cov}_{\text{init}}, \mathcal{D}_\alpha, \tau_{\text{unc}}, \alpha_{\text{unc}} \leftarrow \text{COMPUTECOVERAGE}(\Omega, \mathcal{D}, \mathcal{T}_{\text{train}})$
if $\textit{cov}_{init} = |\mathcal{D}|$ **then**
  return Ω
$\text{cov}_{\text{curr}} \leftarrow \text{cov}_{\text{init}}$
$\Omega' \leftarrow \Omega$
while $\textit{cov}_{curr} \geq \textit{cov}_{init}$ **do**
  $\omega_{\text{new}} \leftarrow \text{INDUCEOPTOCOVER}(\tau_{\text{unc}}, \alpha_{\text{unc}})$
  $\Omega' \leftarrow \text{REMOVEPRECANDDELEFFS}(\Omega') \cup \omega_{\text{new}}$
  $(\mathcal{D}_{\omega_1} \ldots \mathcal{D}_{\omega_m}), (\delta_{\tau_1}, \ldots, \delta_{\tau_j}) \leftarrow \text{PARTITIONDATA}(\Omega', \mathcal{D}, \mathcal{T}_{\text{train}})$
  $\Omega' \leftarrow \text{INDUCEPRECANDDELEFFS}(\Omega', (\mathcal{D}_{\omega_1} \ldots \mathcal{D}_{\omega_m}), (\delta_{\tau_1}, \ldots, \delta_{\tau_j}))$
  $\Omega' \leftarrow \Omega' \cup \text{ENSURENECATOMSSAT}(\omega_{\text{new}}, \mathcal{D}_\alpha)$
  $(\mathcal{D}_{\omega_1} \ldots \mathcal{D}_{\omega_l}), (\delta_{\tau_1}, \ldots, \delta_{\tau_j}) \leftarrow \text{PARTITIONDATA}(\Omega', \mathcal{D}, \mathcal{T}_{\text{train}})$
  $\Omega' \leftarrow \text{INDUCEPRECANDDELEFFS}(\Omega', (\mathcal{D}_{\omega_1} \ldots \mathcal{D}_{\omega_l}), (\delta_{\tau_1}, \ldots, \delta_{\tau_j}))$
  $\text{cov}_{\text{curr}}, \mathcal{D}_\alpha, \tau_{\text{unc}}, \alpha_{\text{unc}} \leftarrow \text{COMPUTECOVERAGE}(\Omega', \mathcal{D}, \mathcal{T}_{\text{train}})$
$\Omega' \leftarrow \text{PRUNENULLDATAOPERATORS}(\Omega')$
return Ω'

Algorithm 5: Pseudocode for our `improve-coverage` successor generator. The inputs are a set of operators Ω, the set of all training demonstrations $\mathcal{D}$, and the corresponding set of trainign tasks $\mathcal{T}_{\text{train}}$. The output is a set of operators Ω' such that $\texttt{coverage}(\Omega') \leq \texttt{coverage}(\Omega)$.

1) `improve-coverage`: Before we discuss the `improve-coverage` generator, we note that the `coverage` term in our hill-climbing objective (Equation III-B1 is computed by using preimage backchaining to find abstract plan suffixes for each demonstration in the training set and record the cumulative number of "uncovered" transitions, that is, $n - k$ for a demonstration of length n and suffix length k.

The pseudocode for the `improve-coverage` successor generator is shown in Algorithm 5. Given the current candidate operator set Ω, training demonstrations $\mathcal{D}$ and corresponding tasks $\mathcal{T}_{\text{train}}$, we first attempt to compute the current coverage of Ω on $\mathcal{D}$. We do this by calling the COMPUTECOVERAGE method. This method simply calls Algorithm 4 on every demonstration $(\overline{x}, \overline{u})$ in $\mathcal{D}$ (the set of objects $\mathcal{O}$ and goal g required by Algorithm 4 are obtained from the training tasks). The COMPUTECOVERAGE method then returns the number of covered transitions[1] (cov_{init}), a dataset of necessary atoms sequences for each demonstration Algorithm 4 is able to cover ($\mathcal{D}_\alpha$), the first uncovered transition encountered ($\tau_{\text{unc}} = (s_k, u_{k+1}, s_{k+1})$), and the corresponding necessary

[1]The total number of transitions in abstract plan suffixes that Algorithm 4 is able to find when run on each demonstration in $\mathcal{D}$.

atoms for the transition (α_{unc}). If the number of covered transitions is the same as the size of the training dataset, then all transitions must be covered and the `coverage` term in our objective (Equation III-B1) must be 0. We thus just return the current operator set Ω with no modifications. Otherwise, we compute a new set of operators Ω' with a lower `coverage` value.

To generate Ω', we first create a new operator with preconditions, add effects and arguments set to cover the transition τ_{unc} and corresponding necessary atoms α_{unc}. The operator's ground controller $\underline{C}(\theta) = u_{k+1}$ is determined directly from the transition's action u_{k+1}. The operator's ground add effects are set to be $\underline{E^+} = (s_{k+1} \setminus s_k) \cap \alpha_{\text{unc}}$. The controller and add effects are lifted by creating a variable v_i for every distinct object that appears in $\underline{C} \cup \underline{E^+}$. The operator's arguments $\overline{v}$ are set to these variables.

Next, we must induce the preconditions and delete effects of this new operator ω_{new}. To this end, we add ω_{new} to our current candidate set, and partition all data in our training set $\mathcal{D}$ into operator specific datasets $\mathcal{D}_\omega$ for each operator ω in our current candidate set. Since operator preconditions and delete effects depend on the partitioning, we first remove these from all operators that are not ω_{new} (REMOVEPRECANDDELEFFS). We perform this partitioning by running the FINDBESTCONSISTENTOP method from Algorithm 4 on this new operator set for every transition in the dataset, though we do not check the condition $s_{i+1}^{pred} \subseteq s_{i+1}$, since the operators do not yet have delete effects specified. While performing this step, we save a mapping δ_{τ_i} from the operator's arguments to the specific objects used to ground it for every transition in the dataset (this will be used for lifting the preconditions and delete effects of each operator below). We assign each transition to the dataset associated with the operator returned by FINDBESTCONSISTENTOP. We return the operator specific datasets $(\mathcal{D}_{\omega_1} \dots \mathcal{D}_{\omega_l})$, as well as the saved object mappings for each transition $(\delta_{\tau_1}, \dots, \delta_{\tau_j})$.

We now induce preconditions and delete effects using $(\mathcal{D}_{\omega_1} \dots \mathcal{D}_{\omega_l})$ and $(\delta_{\tau_1}, \dots, \delta_{\tau_j})$. Before we do this, we delete any operator whose corresponding dataset is empty. Similar to Chitnis et al. [11], we set the *preconditions* to: $P \leftarrow \bigcap_{\tau=(s_i,\cdot,\cdot)\in\mathcal{D}_\omega} \delta_\tau(s_i)$. We also set the *atomic delete effects* to $E_\circ^- \leftarrow \bigcup_{\tau=(s_i,\cdot,s_{i+1})\in\mathcal{D}_\omega} \delta_\tau(s_{i+1}) \setminus \delta_\tau(s_i)$. For every transition (s_i, u_{i+1}, s_{i+1}), let $s_{i+1}^{pred} = (s_i \setminus \underline{E_\circ^-}) \cup \underline{E^+}$. Then, we set $s_{\text{mispred}} = \bigcup_{\tau=(\cdot,\cdot,s_{i+1})\in\mathcal{D}_\omega} s_{i+1}^{pred} \setminus s_{i+1}$. We induce a quantified delete effect for every *predicate* corresponding to atoms in s_{mispred}. We then set each operator's delete effects to be the union of $E_\circ^-$ and the quantified delete effects.

Now that all operators have preconditions and delete effects specified, we must ensure that the newly-added operator (ω_{new}) is able to satisfy the necessary atoms for each of its transitions in $\mathcal{D}_{\omega_{\text{new}}}$. Recall that we set the operator's add effects to be the necessary atoms that changed in the first uncovered transition τ_{unc}. Given the way partitioning is done (specifically the conditions in the FINDBESTCONSISTENTOP method in Algorithm 4), we know that these add effects must satisfy $\alpha_{i+1} \subseteq s_{i+1} \cup \underline{E^+}$ for all transitions $(s_i, u_{i+1}, s_{i+1}) \in \mathcal{D}_{\omega_{\text{new}}}$ with corresponding necessary atoms α_{i+1} for state s_{i+1}. However, the delete effects may cause the necessary atoms to become violated for certain transitions: i.e, $\alpha_{i+1} \nsubseteq (s_{i+1} \setminus \underline{E^-}) \cup \underline{E^+}$. For every such transition, we let $\alpha_{i+1}^{\text{miss}} = \alpha_{i+1} \setminus ((s_{i+1} \setminus \underline{E^-}) \cup \underline{E^+})$. We then create a new operator ω_i^{miss} by copying all components of ω_{new}, and adding lifted atoms from $\alpha_{i+1}^{\text{miss}}$ to both the preconditions and add effects. We modify the operator's arguments to contain new variables accordingly. This now ensures that the necessary atoms are not violated for any transition in $\mathcal{D}_{\omega_{\text{new}}}$. We add these new operators to the current candidate operator set.

After having added new operators to our candidate set in the above step, we must re-partition data and consequently re-induce preconditions and delete effects to match this new partitioning (lines 11-12 of Algorithm 5). We now have a new operator set that is guaranteed to cover the transition τ_{unc} that was initially uncovered. We check whether this new set achieves a lower value for the `coverage` term of our objective, and iterate the above steps until it does.

Finally, after the while loop terminates, we remove all operators from Ω' that have associated datasets that are *empty*. This corresponds exactly to removing operators that are not used in *any* abstract plan suffix computed by COMPUTECOVERAGE and are thus unnecessary for planning.

a) Proof of termination: To see that the main loop of Algorithm 5 is guaranteed to terminate, consider that the operator set Ω' strictly grows larger at every loop iteration (no operators are deleted). Since the predicates are fixed, there is a finite number of possible operators. Thus, at some finite iteration, Ω' will contain every possible operator. At this point, it *must* contain an operator that covers every transition and the loop must terminate.

b) Anytime Removal of Operators with Null Data: In the IMPROVECOVERAGE procedure as illustrated in Algorithm 5, we only prune out operators that do not have any data associated with them after the main while loop has terminated. However, we note here that we can remove such operators from the current operator set (Ω') at any time during the algorithm's loop.

This property arises because *the amount of data associated with a particular operator will only decrease over time*. To see this, note that (1) the number of operators in Ω' only increases over time, and (2) data is assigned to the 'best covering' operator as judged by our heuristic in Equation 2. Given a particular operator ω at some iteration i of the loop, suppose there are d transitions from $\mathcal{D}$ associated with it (i.e, $|\mathcal{D}_\omega| = d$). During future (i.e $> i$) loop iterations, new operators will be added to Ω'. For any of the d transitions in $\mathcal{D}_\omega$, these new operators can either be a worse match (in which case, the transition will remain in $\mathcal{D}_\omega$), or a better match (in which case, the transition will become associated with the new operator). Thus, for any operator ω, once there is no longer any data associated with it, there will *never* be any data associated with it, and it will simply be pruned after the while loop terminates.

As a result, we can prune operators from our current set whenever there is no data associated with them. We do this in our implementation, since it improves our algorithm's wall-clock runtime.

```
REDUCECOMPLEXITY(Ω, D, T_train)
    Ω' ← DELETEOPERATOR(Ω)
    (D_ω1 ... D_ωm), (δ_τ1, ..., δ_τj) ←
      PARTITIONDATA(Ω', D, T_train)
    Ω' ←
      INDUCEPRECANDDELEFFS(Ω', (D_ω1 ... D_ωm),
      (δ_τ1, ..., δ_τj))
    return Ω'
```

Algorithm 6: Pseudocode for our `reduce-complexity` successor generator. The inputs are a set of operators Ω, the set of all training demonstrations $\mathcal{D}$, and the corresponding set of trainign tasks $\mathcal{T}_{\text{train}}$. The output is a set of operators Ω' such that $\texttt{complexity}(\Omega') \leq \texttt{complexity}(\Omega)$.

2) `reduce-complexity`: The pseudocode for our `reduce-complexity` generator is shown in Algorithm 6. As can be seen, the generator is rather simple: we simply delete an operator from the current set (DELETEOPERATOR) and return the remaining operators. Since we've changed the operator set, we must recompute the partitioning and re-induce preconditions and delete effects accordingly.

This generator clearly reduces the `complexity` term from our objective (Equation III-B1), since $|\Omega'| < |\Omega|$.

D. Data Partitioning Heuristic

A key component of our algorithm is the FINDBESTCOVER method from Algorithm 4, which uses a heuristic to associate a transition with an operator when multiple operators satisfy the conditions necessary to 'cover' it. Intuitively, we wish to assign a transition to the operator whose prediction best matches the *observed effects* in the transition. We can do this by simply measuring the discrepancy between the operator's add and delete effects, and the observed add and delete effects in the transition. We make two minor changes to this simple measure that are appropriate to our setting. First, we only use the *atomic* delete effects as part of our measure. We exclude the quantified delete effects because these exist in order to enable our operators to decline to predict particular changes in state. Second, we favor operators that correctly predict which atoms *will not* change. Recall that the ENSURENECATOMSSAT method in Algorithm 5 induces such operators by placing the same atoms in the add effects and preconditions.

Given some transition (s_i, u_{i+1}, s_{i+1}), and some ground operator $\underline{\omega}$ with atomic delete effects $\underline{E_\circ^-}$, our heuristic for data partitioning is represented by the score function shown in equation 2.

$$\begin{aligned}
\underline{\mathcal{K}} &= \underline{E^+} \cap \underline{P} \\
\underline{\mathcal{C}} &= \underline{E^+} \setminus \underline{\mathcal{K}} \\
\text{score} = &|\underline{\mathcal{C}} \setminus (s_{i+1} \setminus s_i)| + \\
&|(s_{i+1} \setminus s_i) \setminus \mathcal{C}| + \\
&|(\underline{E_\circ^-} \setminus (s_i \setminus s_{i+1}))| + \\
&|(s_i \setminus s_{i+1}) \setminus \underline{E_\circ^-}| - \underline{\mathcal{C}}
\end{aligned} \tag{2}$$

Once all eligible operators have been scored, we simply pick the lowest-scoring operator to associate with this transition. If multiple operators achieve the same score, we break ties arbitrarily.

E. Learning Samplers

In addition to operators, we must also learn samplers to propose continuous parameters for controllers during plan refinement. We directly adapt existing approaches [11, 37] to accomplish this and learn one sampler per operator of the following form: $\sigma(x, o_1, \ldots, o_k) = s_\sigma(x[o_1] \oplus \cdots \oplus x[o_k])$, where $x[o]$ denotes the feature vector for o in x, the $\oplus$ denotes concatenation, and s_σ is the model to be learned. Specifically, we treat the problem as one of supervised learning on each of the datasets associated with each operator: $\mathcal{D}_\omega$. Recall that for every transition (x_i, u_{i+1}, x_{i+1}) in $\mathcal{D}_\omega$, we save a mapping $\delta : \overline{v} \to \mathcal{O}_\tau$ from the operator's arguments $\overline{v}$ to objects to ground the operator with. Recall also that every action is a hybrid controller with discrete parameters and continuous parameters θ. To create a datapoint that can be used for supervised learning for the associated sampler, we can reuse this substitution to create an input vector $x[\delta_\tau(v_1)] \oplus \cdots \oplus x[\delta(v_k)]$, where $(v_1, \ldots, v_k) = \overline{v}$. The corresponding output for supervised learning is the continuous parameter vector θ in the action u_{i+1}.

Following previous work by Silver et al. [37] and Chitnis et al. [11], we learn two neural networks to parameterize each sampler. The first neural network takes in $x[o_1] \oplus \cdots \oplus x[o_k]$ and regresses to the mean and covariance matrix of a Gaussian distribution over θ. We assume that the desired distribution has nonzero measure, but the covariances can be arbitrarily small in practice. To improve the representational capacity of this network, we learn a second neural network that takes in $x[o_1] \oplus \cdots \oplus x[o_k]$ *and* θ, and returns true or false. This classifier is then used to rejection sample from the first network. To create negative examples, we use all transitions such that the controller used in the transition matches the current controller, but the transition is not in the operator's dataset $\mathcal{D}_\omega$.

F. Additional Experiment Details

Here we provide detailed descriptions of each of experiment environments. See the accompanying code for implementations.

1) Screws Environment: An environment where the agent controls a crane to pick up screws and place them in a receptacle.

- **Types:**

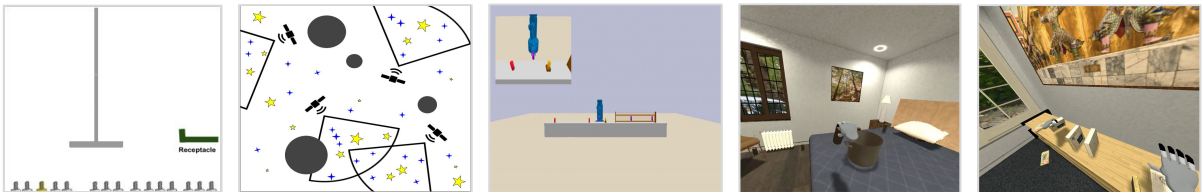

Fig. 2. **Environments**. Visualizations for *Screws*, *Satellites*, *Painting*, *Collecting Cans*, and *Sorting Books*.

  - The `screw` type has features `x`, `y`, `held`.
  - The `receptacle` type has features `x`, `y`.
  - The `gripper` type has features `x`, `y`.
- **Predicates:** `Pickable(?x0:gripper, ?x1:receptacle)`, `AboveReceptacle(?x0:gripper, ?x1:receptacle)`, `HoldingScrew(?x0:gripper, ?x1:screw)`, `ScrewInReceptacle(?x0:screw, ?x1:receptacle)`.
- **Actions:**
  - `MoveToScrew(?x0: gripper, ?x1: screw)`: moves the gripper to be `Near` the screw ?x1.
  - `MoveToReceptacle(?x0: gripper, ?x1: receptacle)`: moves the gripper to be AboveReceptacle(?x0:gripper, ?x1:receptacle)
  - `MagnetizeGripper(?x0: gripper)`: Magnetizes the gripper at the current location, which causes all screws that the gripper is `Near` to be held by the gripper.
  - `DemagnetizeGripper(?x0: gripper)`: Demagnetizes the gripper at the current location, which causes all screws that are being held by the gripper to fall.
- **Goal:** The agent must make `ScrewInReceptacle(?x0:screw, ?x1:receptacle)` true for a particular screw that varies per task.

2) Cluttered 1D Environment: A simple environment where the robot must move and collect objects cluttered along a 1D line. An object can only be collected if it is reachable.

- **Types:**
  - The `robot` type has features `x`.
  - The `dot` type has features `x`, `grasped`.
- **Predicates:** `NextTo(?x0:robot, ?x1:dot)`, `NextToNothing(?x0:robot)`, `Grasped(?x0:robot, ?x1:dot)`.
- **Actions:**
  - `MoveGrasp(?x0: robot, ?x1: dot, [move_or_grasp, x])`: A single controller that performs both moving and grasping. If `move_or_grasp` < 0.5, then the controller moves the robot to a continuous position `y`. Else, the controller grasps the dot ?x1 if it is within range.
- **Goal:** The agent must make `Grasped(?x0:robot, ?x1:dot)` true for a particular set of dots that varies per task.

3) Satellites Environment: A 2D environment inspired by the Satellites planning problem from Bacchus [7]. Note that we use two different variants of this environment. In "Satellites Simple", there is only ever one object in the domain, and a reading must be taken from that object. In the full 'Satellites" environment, there are multiple objects strewn throughout the domain, and readings must be taken from each of these.

- **Types:**
  - The `satellite` type has features `x`, `y`, `theta`, `instrument`, `calibration_obj_id`, `is_calibrated`, `read_obj_id`, `shoots_chem_x`, `shoots_chem_y`.
  - The `object` type has features `id`, `x`, `y`, `has_chem_x`, `has_chem_y`.
- **Predicates:** `Sees(?x0:satellite, ?x1:object)`, `CalibrationTarget(?x0:satellite, ?x1:object)`, `IsCalibrated(?x0:satellite)`, `HasCamera(?x0:satellite)`, `HasInfrared(?x0:satellite)`, `HasGeiger(?x0:satellite)`, `ShootsChemX(?x0:satellite)`, `ShootsChemY(?x0:satellite)`, `HasChemX(?x0:satellite)`, `HasChemY(?x0:satellite)`, `CameraReadingTaken(?x0:satellite, ?x1:object)`, `InfraredReadingTaken(?x0:satellite, ?x1:object)`, `GeigerReadingTaken(?x0:satellite, ?x1:object)`.
- **Actions:**
  - `MoveTo(?x0:satellite, ?x1:object, [x, y])`: Moves the satellite `?x0` to be at `x, y`.
  - `Calibrate(?x0:satellite, ?x1:object)`: Tries to calibrate the satellite `?x0` against object `?x1`. This will only succeed (i.e, make `IsCalibrated(?x0:satellite)` true) if `?x1` is the calibration target of `?x0`.
  - `ShootChemX(?x0:satellite, ?x1: object)`: Tries to shoot a pellet of chemical X from satellite ?x0. This will only succeed if `?x0`

both has chemical X and is capable of shooting it.
  - `ShootChemY(?x0:satellite, ?x1:object)`: Tries to shoot a pellet of chemical Y from satellite ?x0. This will only succeed if `?x0` both has chemical Y and is capable of shooting it.
  - `UseInstrument(?x0:satellite, ?x1:object)`: Tries to use the instrument possessed by `?x0` on object `?x1` (note that we assume ?x0 only possesses a single instrument).
- **Goal:** The agent must take particular readings (i.e some combination of `CameraReadingTaken(?x0:satellite, ?x1:object)`, `InfraredReadingTaken(?x0:satellite, ?x1:object)`, `GeigerReadingTaken(?x0:satellite, ?x1:object)`) from a specific set of objects that varies per task.

4) Painting Environment: A challenging robotics environment used by Silver et al. [35, 37]. A robot in 3D must pick, wash, dry, paint, and then place various objects.

- **Types:**
  - The `object` type has features `x`, `y`, `z`, `dirtiness`, `wetness`, `color`, `grasp`, `held`.
  - The `box` type has features `x`, `y`, `color`.
  - The `lid` type has features `open`.
  - The `shelf` type has features `x`, `y`, `color`.
  - The `robot` type has features `x`, `y`, `fingers`.
- **Predicates:** `InBox(?x0:obj)`, `InShelf(?x0:obj)`, `IsBoxColor(?x0:obj, ?x1:box)`, `IsShelfColor(?x0:obj, ?x1:shelf)`, `GripperOpen(?x0:robot)`, `OnTable(?x0:obj)`, `NotOnTable(?x0:obj)`, `HoldingTop(?x0:obj)`, `HoldingSide(?x0:obj)`, `Holding(?x0:obj)`, `IsWet(?x0:obj)`, `IsDry(?x0:obj)`, `IsDirty(?x0:obj)`, `IsClean(?x0:obj)`.
- **Actions:**
  - `Pick(?x0:robot, ?x1:obj, [grasp])`: picks up a particular object, if grasp $>$ 0.5 it performs a top grasp otherwise a side grasp.
  - `Wash(?x0:robot)`: washes the object in hand, which is needed to clean the object.
  - `Dry(?x0:robot)`: drys the object in hand, which is needed after you wash the object.
  - `Paint(?x0:robot, [color])`: paints the object in hand a particular color specified by the continuous parameter.
  - `Place(?x0:robot, [x, y, z])`: places the object in hand at a particular x, y, z location specified by the continuous parameters.
  - `OpenLid(?x0:robot, ?x1:lid)`: opens a specific lid, which is need to place objects inside the box.
- **Goal:** A robot in 3D must pick, wash, dry, paint, and then place various objects in order to get `InBox(?x0:obj)` and `IsBoxColor(?x0:obj, ?x1:box)`, or `InShelf(?x0:obj)` and `IsShelfColor(?x0:obj, ?x1:shelf)` true for particular goal objects.

5) Cluttered Painting Environment: Same as in Painting, except that the robot can be next to many objects at a time.

- **Types:**
  - The `object` type has features `x`, `y`, `z`, `dirtiness`, `wetness`, `color`, `grasp`, `held`.
  - The `box` type has features `x`, `y`, `color`.
  - The `lid` type has features `open`.
  - The `shelf` type has features `x`, `y`, `color`.
  - The `robot` type has features `x`, `y`, `fingers`.
- **Predicates:** `InBox(?x0:obj)`, `InShelf(?x0:obj)`, `IsBoxColor(?x0:obj, ?x1:box)`, `IsShelfColor(?x0:obj, ?x1:shelf)`, `GripperOpen(?x0:robot)`, `OnTable(?x0:obj)`, `NotOnTable(?x0:obj)`, `HoldingTop(?x0:obj)`, `HoldingSide(?x0:obj)`, `Holding(?x0:obj)`, `IsWet(?x0:obj)`, `IsDry(?x0:obj)`, `IsDirty(?x0:obj)`, `IsClean(?x0:obj)`, along with RepeatedNextTo Predicates: `NextTo(?x0:robot, ?x1:obj)`, `NextToBox(?x0:robot, ?x1:box)`, `NextToShelf(?x0:robot, ?x1:shelf)`, `NextToTable(?x0:robot, ?x1:table)`.
- **Actions:**
  - `Pick(?x0:robot, ?x1:obj, [grasp])`: picks up a particular object, if grasp ¿ 0.5 it performs a top grasp otherwise a side grasp.
  - `Wash(?x0:robot)`: washes the object in hand, which is needed to clean the object.
  - `Dry(?x0:robot)`: drys the object in hand, which is needed after you wash the object.
  - `Paint(?x0:robot, [color])`: paints the object in hand a particular color specified by the continuous parameter.
  - `Place(?x0:robot, [x, y, z])`: places the object in hand at a particular x, y, z location specified by the continuous parameters.
  - `OpenLid(?x0:robot, ?x1:lid)`: opens a specific lid, which is need to place objects inside the box.
  - `MoveToObj(?x0:robot, ?x1:obj, [x])`: moves to a particular object with certain displacement x.
  - `MoveToBox(?x0:robot, ?x1:box, [x])`: moves to a particular box with certain displacement x.
  - `MoveToShelf(?x0:robot, ?x1:shelf, [x])`: moves to a particular shelf with certain displacement x.
- **Goal:** A robot in 3D must pick, wash, dry, paint,

and then place various objects in order to get `InBox(?x0:obj)` and `IsBoxColor(?x0:obj, ?x1:box)`, or `InShelf(?x0:obj)` and `IsShelfColor(?x0:obj, ?x1:shelf)` true for particular goal objects. In contrast to the previous painting environment, we also need to navigate to the right objects (i.e. all objects are not always reachable from any states). This version of the environment requires operators with ignore effects.

6) BEHAVIOR Environment Details: A set of complex, long-horizon household robotic tasks simulated with realistic 3D models of objects and homes [38]. In *Opening Presents*, the robot must open a number of boxes. In *Locking Windows*, the robot must close a number of open windows. In *Collecting Cans*, the robot must pick up a number of empty soda cans strewn amongst the house and throw them into a trash can. In *Sorting Books*, the robot must find books in a living room and place them each onto a cluttered shelf.

- **Types:**
  - Many `object` types that range from relevant types like `hardbacks` and `notebooks` to many irrelevant types like `toys` and `jars`. All `object` types have features from `location` and `orientation` to `graspable` and `open`. For a complete list of `object` types and features see [38].
- **Predicates:** `Inside(?x0:obj, ?x1:obj)`, `OnTop(?x0:obj, ?x1:obj)`, `Reachable-Nothing()`, `HandEmpty()`, `Holding(?x0:obj)`, `Reachable()`, `Openable(?x0:obj)`, `Not-Openable(?x0:obj)`, `Open(?x0:obj)`, `Closed(?x0:obj)`.
- **Actions:**
  - `NavigateTo(?x0:obj)`: navigates to make a particular object reachable.
  - `Grasp(?x0:obj, [x, y, z])`: picks up a particular object with the hand starting at a particular relative x, y, z location specified by the continuous parameters.
  - `PlaceOnTop(?x0:obj)`: places the object in hand ontop of another object as long as the agent is holding an object and is in range of the object to be placed onto.
  - `PlaceInside(?x0:obj)`: places the object in hand inside another object as long as the agent is holding an object and is in range of the object to be placed into.
  - `Open(?x0:obj))`: opens a specific object (windows, doors, boxes, etc.) if it is 'openable'.
  - `Close(?x0:obj))`: closes a specific object (windows, doors, boxes, etc.) if it is currently in an 'open' state.
- **Goal:** In *Opening Presents*, the robot must `Open(?x0:package)` a number of boxes of type `package` around the room. In *Locking Windows*, the robot must navigate around the house to `Close(?x0:window)` a number of windows. In *Collecting Cans*, the robot must pick up a number of empty soda cans of type `pop` strewn amongst the house and throw them into a trash can of type `bucket`. This will satisfy the goal of getting `Inside(?x0:pop, ?x1:bucket)` for every soda can around the house. In *Sorting Books*, the robot must find books of type `hardback` and `notebook` in a living room and place them each onto a cluttered shelf (i.e. satisfy the goal of `OnTop(?x0:hardback, ?x1:shelf)` and `OnTop(?x0:notebook, ?x1:shelf)` for a number of books).

G. Additional Approach Details

Here we provide detailed descriptions of each approach evaluated in experiments. For the approaches that learn operators, we use A^* search with the lmcut heuristic [18] as the high-level planner for bilevel planning in non-BEHAVIOR environments, and use Fast Downward [17] in a configuration with minor differences from `lama-first` as the high-level planner in BEHAVIOR environments, since A^* search was unable to find abstract plans given the large state and action spaces of these tasks. All approaches also iteratively resample until the simulator f verifies that the transition has been achieved, except for GNN Model-Free, which is completely model-free. See Section IV-A for high-level descriptions and the accompanying code for implementations.

1) Ours:

- **Operator Learning:** We learn operators via the hill-climbing search described in Section III-B. For our objective (Equation III-B1), we set the λ term to be $1/|\mathcal{D}|$, where $|\mathcal{D}|$ represents the number of transitions in the training demonstrations.
- **Sampler Learning:** As described in Section VII-E, each sampler consists of two neural networks: a generator and a discriminator. The generator outputs the mean and diagonal covariance of a Gaussian, using an exponential linear unit (ELU) to assure PSD covariance. The generator is a fully-connected neural network with two hidden layers of size 32, trained with Adam for 50,000 epochs with a learning rate of $1\mathrm{e}{-3}$ using Gaussian negative log likelihood loss. The discriminator is a binary classifier of samples output by the generator. Negative examples for the discriminator are collected from other skill datasets. The classifier is a fully-connected neural network with two hidden layers of size 32, trained with Adam for 10,000 epochs with a learning rate of $1\mathrm{e}{-3}$ using binary cross entropy loss. During planning, the generator is rejection sampled using the discriminator for up to 100 tries, after which the last sample is returned.
- **Planning:** The number of abstract plans for high-level planning was set to $N_{\text{abstract}} = 8$ for our non-BEHAVIOR domains, and $N_{\text{abstract}} = 1$ for our BEHAVIOR domains. The samples per step for refinement was set to $N_{\text{samples}} = 10$ for all environments.

2) Cluster and Intersect:: This is the operator learning approach used by Silver et al. [37].

- **Operator Learning:** This approach learns STRIPS operators by attempting to induce a different operator for every set of unique lifted effects (See Silver et al. [37] for more information).
- **Sampler Learning and Planning:** Same as Ours (See Section VII-G1 for more details).

3) LOFT:: This is the operator learning approach used by Silver et al. [35]. We include a version ('LOFT+Replay') that is allowed to mine additional negative data from the environment to match the implementation of the original authors. We also include a version ('LOFT') that is restricted to learning purely from the demonstration data.

- **Operator Learning:** This approach learns operators similar to the Cluster and Intersect baseline, except that it uses search to see if it can modify the operators after performing Cluster and Intersect (See Silver et al. [35] for more information).
- **Sampler Learning and Planning:** Same as Ours (See Section VII-G1 for more details).

4) CI + QE:: A baseline variant of Cluster and Intersect that is capable of learning operators that have quantified delete effects in addition to atomic delete effects.

- **Operator Learning:** This approach first runs Cluster and Intersect, then attempts to induce ignore effects by performing a hill-climbing search over possible choices of ignore effects using prediction error as the metric to be optimized.
- **Sampler Learning and Planning:** Same as Ours (See Section VII-G1 for more details).

5) GNN Shooting:: This approach trains a graph neural network (GNN) [8] policy. This GNN takes in the current state x, abstract state $s = \textsc{Abstract}(x, \Psi_G)$, and goal g. It outputs an action via a one-hot vector over $\mathcal{C}$ corresponding to which controller to execute, one-hot vectors over all objects at each discrete argument position, and a vector of continuous arguments. We train the GNN using behavior cloning on the dataset $\mathcal{D}$. At evaluation time, we sample trajectories by treating the GNN's output continuous arguments as the mean of a Gaussian with fixed variance. We use the known transition model f to check if the goal is achieved, and repeat until the planning timeout is reached.

- **Planning:** Repeat until the goal is reached: query the model on the current state, abstract state, and goal to get a ground skill. Invoke the ground skill's sampler up to 100 times to find a subgoal that leads to the abstract successor state predicted by the skill's operator. If successful, simulate the state forward; otherwise, terminate with failure.
- **Learning:** This approach essentially learns a TAMP planner in the form of a GNN. Following the baselines presented in prior work [11], the GNN is a standard encode-process-decode architecture with 3 message passing steps. Node and edge modules are fully-connected neural networks with two hidden layers of size 16. We follow the method of Chitnis et al. [11] for encoding object-centric states, abstract states, and goals into graph inputs. To get graph outputs, we use node features to identify the object arguments for the skill and a global node with a one-hot vector to identify the skill identity. The models are trained with Adam for 1000 epochs with a learning rate of $1\mathrm{e}{-3}$ and batch size 128 using MSE loss.

6) GNN Model-Free:: A baseline that uses the same trained GNN as above, but at evaluation time, directly executes the policy instead of checking execution using f. This has the advantage of being more efficient to evaluate than GNN Shooting, but is less effective.

H. Additional Experimental Results and Analyses

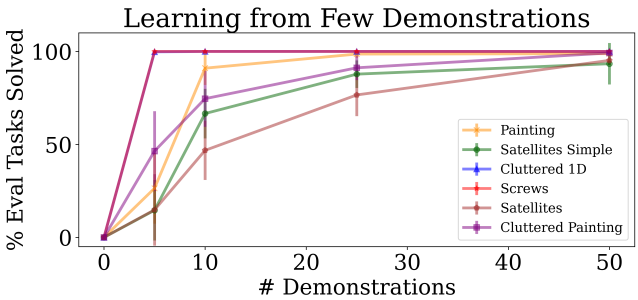

Fig. 3. Data-efficiency of main approach.

1) Data efficiency of our learning approach: Figure 3 shows our method's testing success rate as a function of the size of its training set for our non-BEHAVIOR environments. Our approach's performance improves with more data, though as the dataset size increases, the impact of additional data on performance reduces.

Table III shows the number of operators learned for all operator learning methods in all domains. Our approach learns the lowest number of operator sets across all environments and massively out performs other approaches on this metric in *Collecting Cans*, and *Sorting Books*. These results further highlight our ability to learn operator sets that efficient for high-level planning, but also simpler and therefore, more likely to generalize to new environments.

2) Comparisons against baselines: We have already established that our approach learns operators that lead to more effective *bilevel planning* than baselines. In this section, we are interested in comparing our approach with baselines on three additional metrics: (1) the efficiency of high-level planning using learned operators, (2) the efficiency of the learning algorithm itself, (3) the simplicity of operator sets we learn.

Figure 4 shows the nodes created during high-level planning for each of our various environments and operator learning methods. We can see that operators learned by our approach generally lead to comparable or fewer node creations during planning when compared to baselines. In many of the environments where baseline methods are able to achieve a number of points with fewer node creations — *Cluttered 1D*,

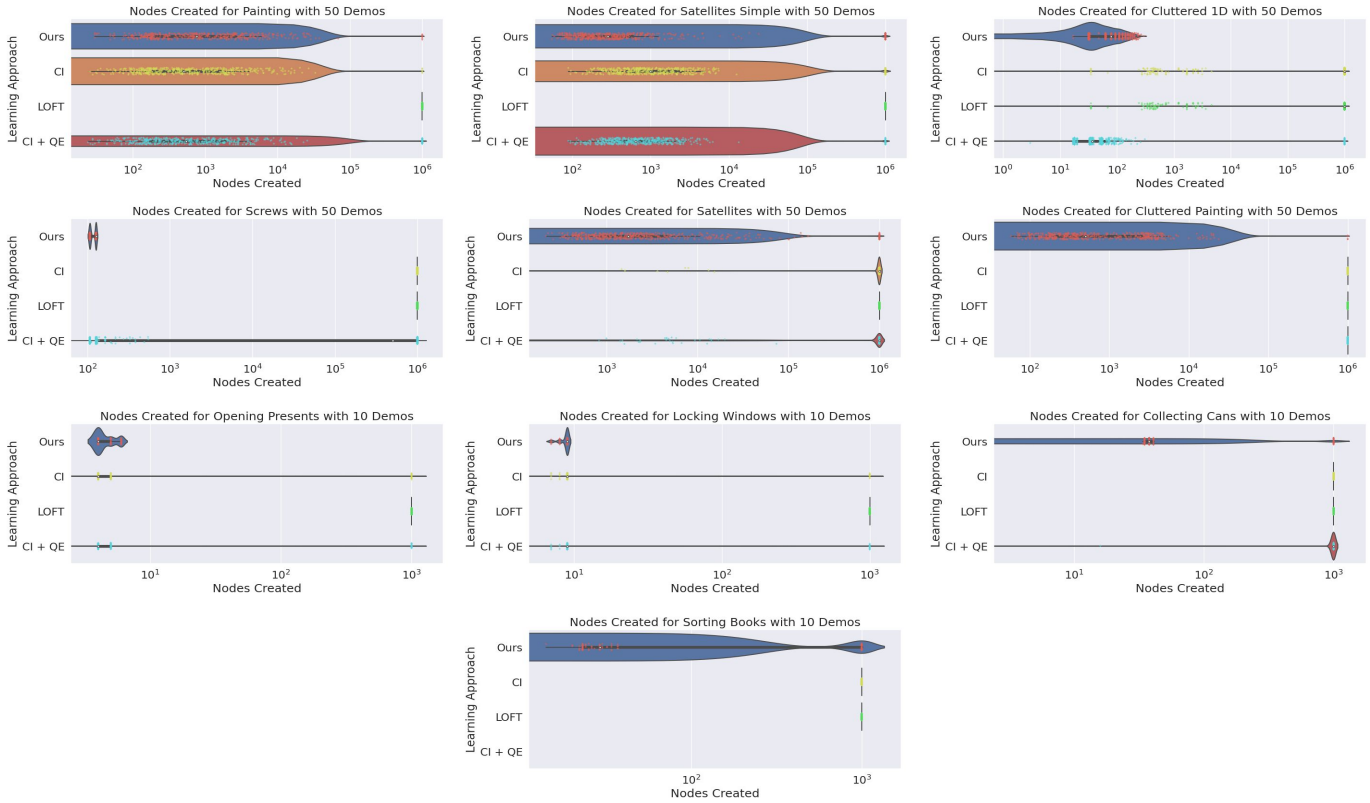

Fig. 4. **Nodes Created by Operator Learning Approaches**. We show scatter plots of the nodes created (x-axis) for each operator learning approach (y-axis). We also include a violin graph to visualize the density of points throughout the graph. If bilevel planning failed, we set the nodes created to 10^6 for non-BEHAVIOR domains and 10^3 for BEHAVIOR domains. Our approach achieves a low number of nodes created across when compared to baselines in most domains.

Environment	Ours	LOFT	LOFT+replay	CI	CI + QE	GNN
Painting	**69.35 (3.58)**	92.26 (11.41)	135.73 (6.45)	**70.95 (5.07)**	**67.08 (5.86)**	2220.19 (181.29)
Satellites Simple	**19.38 (7.83)**	52.73 (18.35)	438.44 (51.62)	23.29 (5.38)	**15.96 (4.70)**	1625.69 (218.88)
Clutter 1D	**17.98 (1.06)**	68.04 (17.68)	366.89 (146.09)	62.68 (14.89)	28.58 (3.68)	1164.92 (84.74)
Screws	1.31 (0.04)	143.60 (49.10)	5712.80 (736.84)	**0.32 (0.02)**	708.98 (1023.02)	1369.59 (68.44)
Satellites	**16.12 (0.55)**	353.67 (52.78)	902.99 (148.22)	107.04 (11.94)	87.24 (10.49)	3043.62 (285.27)
Cluttered Painting	**131.68 (5.05)**	1699.52 (216.71)	7364.03 (532.67)	470.32 (40.38)	2788.74 (1330.38)	4615.70 (334.11)
Opening Presents	**28.91 (11.26)**	106.57 (27.72)	-	100.62 (23.66)	92.63 (17.01)	185.53 (6.63)
Locking Windows	**16.77 (1.55)**	62.55 (10.12)	-	61.71 (8.95)	45.51 (5.74)	319.09 (7.61)
Collecting Cans	3728.73 (9544.75)	1520.93 (354.20)	-	**576.89 (100.57)**	781.38 (350.46)	2121.86 (120.51)
Sorting Books	4981.79 (14460.37)	6423.03 (602.44)	-	**1528.18 (111.18)**	-	5359.99 (170.46)

TABLE II
LEARNING TIMES IN SECONDS ON TRAINING DATA FOR ALL DOMAINS. NOTE THAT BEHAVIOR DOMAINS (BOTTOM 4) USE TRAINING SET SIZES OF 10 TASKS, WHILE ALL OTHER DOMAINS USE TRAINING AND TESTING SET SIZES OF 50 TASKS. THE STANDARD DEVIATION IS SHOWN IN PARENTHESES.

Opening Presents, and *Locking Windows* — our method has a significantly higher success rate.

Table II shows the learning times for all methods in all domains[2]. Our approach achieves the lowest learning time in 7/10 domains. Upon inspection of our method's performance on the 'Locking Windows' and 'Collecting Cans' domains, we discovered that the high average learning times are because of a few outlier seeds encountering local minima learning, yielding large and complex operator sets (this is the reason for the extremely high standard deviation).

[2]Note that there is no entry for 'CI + QE' for sorting books because learning exceeded the memory limit of our hardware (192 GB)

I. Learned Operator Examples

Finally, we provide operator examples to demonstrate our approaches ability to overcome overfiting to specific situations. Figure 5 shows a comparison of the operators learned with *Open* in *Opening Packages* environment and *NavigateTo* in *Collecting Cans* environment across our approach and 'CI + QE' (the most competitive baseline in these environments). As shown, by optimizing prediction error 'CI + QE' learns a

Environment	Ours	LOFT	LOFT+replay	CI	CI + QE
Painting	**10.00 (0.00)**	13.60 (0.80)	19.20 (0.39)	11.00 (0.00)	10.20 (0.40)
Satellites Simple	**7.40 (0.79)**	10.90 (1.44)	33.80 (3.70)	10.40 (1.20)	9.30 (0.9)
Clutter 1D	**2.00 (0.00)**	7.10 (1.64)	16.10 (2.11)	7.10 (1.64)	3.00 (0.44)
Screws	**4.0 (0.00)**	14.80 (1.98)	91.14 (5.11)	14.80 (1.98)	4.80 (0.97)
Satellites	**7.00 (0.00)**	19.80 (2.60)	59.60 (4.45)	16.30 (1.10)	13.9 (0.83)
Cluttered Painting	**13.00 (0.00)**	28.00 (0.00)	157.8 (6.49)	25.20 (2.31)	20.70 (1.61)
Opening Presents	**2.30 (0.90)**	10.80 (2.99)	-	10.80 (2.99)	9.80 (1.83)
Locking Windows	**2.00 (0.00)**	6.10 (0.70)	-	6.10 (0.70)	4.70 (0.64)
Collecting Cans	**6.10 (5.37)**	57.40 (9.43)	-	52.90 (8.41)	13.40 (2.33)
Sorting Books	**14.70 (7.57)**	76.70 (5.62)	-	75.80 (5.79)	-

TABLE III
AVERAGE NUMBER OF OPERATORS LEARNED FOR ALL DOMAINS. NOTE THAT BEHAVIOR DOMAINS (BOTTOM 4) USE TRAINING SET SIZES OF 10 TASKS, WHILE ALL OTHER DOMAINS USE TRAINING AND TESTING SET SIZES OF 50 TASKS. THE STANDARD DEVIATION IS SHOWN IN PARENTHESES.

number of operators to describe the same amount of transitions that is covered by the single operator our approach learns. Upon inspection, ‘CI + QE‘ learns overly specific operators when trying to cluster effects that try to predict the entire state to the point where ‘Quantified Delete Effects’ are not fully utilized. For the full set of operators learned by our algorithm on the ‘Sorting Books’ task, see Figure 6.

```
Open-package0:
    Arguments: [?x0:package]
    Preconditions: [
        closed-package(?x0:package),
        handempty(),
        openable-package(?x0:package),
        reachable-package(?x0:package)]
    Add Effects: [open-package(?x0:package)]
    Delete Effects: [closed-package(?x0:package)]
    Quantified Delete Effects:
        [ontop-package-room_floor]
    Controller: Open-package(?x0:package),
```

```
Open-package0:
    Arguments: [?x0:package]
    Preconditions: [
        closed-package(?x0:package),
        handempty(),
        openable-package(?x0:package),
        reachable-package(?x0:package)]
    Add Effects: [open-package(?x0:package)]
    Delete Effects: [closed-package(?x0:package)]
    Quantified Delete Effects: []
    Controller: Open-package(?x0:package),

Open-package1:
    Arguments: [?x0:room_floor, ?x1:package]
    Preconditions: [
        closed-package(?x1:package),
        handempty(),
        not-openable-room_floor(?x0:room_floor),
        ontop-package-room_floor(?x1:package, ?x0:room_floor),
        openable-package(?x1:package),
        reachable-package(?x1:package),
        reachable-room_floor(?x0:room_floor)]
    Add Effects: [open-package(?x1:package)]
    Delete Effects: [closed-package(?x1:package),
        ontop-package-room_floor(?x1:package, ?x0:room_floor)]
    Quantified Delete Effects: []
    Controller: Open-package(?x1:package)}
```

```
NavigateTo-pop0:
    Arguments: [?x0:pop]
    Preconditions: [
        handempty(),
        not-openable-pop(?x0:pop)]
    Add Effects: [reachable-pop(?x0:pop)]
    Delete Effects: []
    Quantified Delete Effects: [
        ontop-pop-pop,
        reachable-bed,
        reachable-bucket,
        reachable-pop]
    Controller: NavigateTo-pop(?x0:pop)}
```

```
NavigateTo-pop0:
    Arguments: [?x0:bed, ?x1:pop]
    Preconditions: [
        handempty(),
        not-openable-bed(?x0:bed),
        not-openable-pop(?x1:pop),
        ontop-pop-bed(?x1:pop, ?x0:bed),
        reachable-bed(?x0:bed)]
    Add Effects: [reachable-pop(?x1:pop)]
    Delete Effects: [reachable-bed(?x0:bed)]
    Quantified Delete Effects: []
    Controller: NavigateTo-pop(?x1:pop),

NavigateTo-pop1:
    Arguments: [?x0:pop]
    Preconditions: [
        handempty(),
        not-openable-pop(?x0:pop),
        reachable-pop(?x0:pop)]
    Add Effects: []
    Delete Effects: []
    Quantified Delete Effects: []
    Controller: NavigateTo-pop(?x0:pop),

NavigateTo-pop2:
    Arguments: [?x0:pop]
    Preconditions: [
        handempty(),
        not-openable-pop(?x0:pop)]
    Add Effects: [reachable-pop(?x0:pop)]
    Delete Effects: []
    Quantified Delete Effects: []
    Controller: NavigateTo-pop(?x0:pop),
```

Fig. 5. **Operator Comparison**. Operators learned after our approach (left) and ‘CI+QE‘ (right), for *Open* in *Opening Packages* environment (top) and *NavigateTo* in *Collecting Cans* Cans environment. Our approach learns fewer operators that are generally simpler, and thus more conducive to effective bilevel planning and generalization.

Grasp-notebook0:
Arguments: [?x0:notebook]
Preconditions: [handempty(), not-openable-notebook(?x0:notebook), reachable-notebook(?x0:notebook)]
Add Effects: [holding-notebook(?x0:notebook)]
Delete Effects: [handempty(), reachable-notebook(?x0:notebook)]
Quantified Delete Effects: [ontop-notebook-coffee_table, ontop-notebook-room_floor]
Controller: Grasp-notebook(?x0:notebook)

NavigateTo-notebook0:
Arguments: [?x0:notebook]
Preconditions: [handempty(), not-openable-notebook(?x0:notebook)]
Add Effects: [reachable-notebook(?x0:notebook)]
Delete Effects: []
Quantified Delete Effects: [reachable-board_game, reachable-coffee_table, reachable-hardback, reachable-notebook, reachable-shelf, reachable-video_game]
Controller: NavigateTo-notebook(?x0:notebook)

PlaceOnTop-shelf0:
Arguments: [?x0:shelf, ?x1:hardback]
Preconditions: [holding-hardback(?x1:hardback), not-openable-hardback(?x1:hardback), not-openable-shelf(?x0:shelf), reachable-shelf(?x0:shelf)]
Add Effects: [handempty(), ontop-hardback-shelf(?x1:hardback, ?x0:shelf)]
Delete Effects: [holding-hardback(?x1:hardback)]
Quantified Delete Effects: []
Controller: PlaceOnTop-shelf(?x0:shelf)

PlaceOnTop-shelf1:
Arguments: [?x0:shelf, ?x1:notebook]
Preconditions: [holding-notebook(?x1:notebook), not-openable-notebook(?x1:notebook), not-openable-shelf(?x0:shelf), reachable-shelf(?x0:shelf)]
Add Effects: [handempty(), ontop-notebook-shelf(?x1:notebook, ?x0:shelf)]
Delete Effects: [holding-notebook(?x1:notebook)]
Quantified Delete Effects: []
Controller: PlaceOnTop-shelf(?x0:shelf)

Grasp-hardback0:
Arguments: [?x0:hardback]
Preconditions: [handempty(), not-openable-hardback(?x0:hardback), reachable-hardback(?x0:hardback)]
Add Effects: [holding-hardback(?x0:hardback)]
Delete Effects: [handempty(), reachable-hardback(?x0:hardback)]
Quantified Delete Effects: [ontop-hardback-coffee_table, ontop-hardback-room_floor]
Controller: Grasp-hardback(?x0:hardback)

NavigateTo-shelf0:
Arguments: [?x0:shelf]
Preconditions: [not-openable-shelf(?x0:shelf)]
Add Effects: [reachable-shelf(?x0:shelf)]
Delete Effects: []
Quantified Delete Effects: [ontop-hardback-coffee_table, reachable-board_game, reachable-coffee_table, reachable-hardback, reachable-notebook, reachable-video_game]
Controller: NavigateTo-shelf(?x0:shelf)

NavigateTo-hardback0:
Arguments: [?x0:hardback]
Preconditions: [handempty(), not-openable-hardback(?x0:hardback)]
Add Effects: [reachable-hardback(?x0:hardback)]
Delete Effects: []
Quantified Delete Effects: [reachable-board_game, reachable-coffee_table, reachable-hardback, reachable-notebook, reachable-shelf, reachable-video_game]
Controller: NavigateTo-hardback(?x0:hardback)

Fig. 6. Sorting Books learned operators.